Systems Support for Ubiquitous Computing: A Case Study of two Implementations of Labscape

Larry Arnstein[1] Robert Grimm[1], Chia-Yang Hung[1], Jong Hee Kang[1], Anthony LaMarca[2], Stefan B. Sigurdsson[2], Jing Su[2], Gaetano Borriello[1,2]

[1] Department of Computer Science & Engineering, University of Washington, Sieg Hall, Seattle, Washington, 98195, USA
[2] Intel Research laboratory @ Seattle, 1100 NE 45th St. Seattle Washington, 98105, USA

Abstract

Labscape, a ubiquitous computing environment for cell biologists, was implemented twice: once using only standard tools for distributed systems (TCP sockets, and shared file systems) and once using *one.world*, a runtime system designed specifically to support ubiquitous applications. The application is analyzed in terms of the system properties that are required to provide a fluid user experience. Though the two implementations are functionally and architecturally similar, we found a significant difference in the degree to which they each exhibited the required properties. The fact that *one.world* was not designed specifically with Labscape in mind suggests that ubiquitous applications have many requirements in common, and therefore can benefit from a system support layer for coping with dynamic environments. We present, in detail, the concepts embodied in *one.world* that we have found to be most important for Labscape, and how some of these concepts might be extended even further.

1. Introduction

Labscape is a ubiquitous laboratory assistant that helps cell biologists in several ways: it presents needed information in the context of the experiment; it records experiment data and observations as the work is performed; and it provides ubiquitous access to the experiment record in support of communication and collaboration. This project was not undertaken explicitly to evaluate systems technologies for ubiquitous computing, yet we have learned a great deal about the role that systems technology should play in realizing high performance, robust smart environments.

Labscape was initially developed using standard tools for distributed systems (TCP sockets, and shared file systems). Although we achieved some degree of success in our development effort, the resulting system failed to deliver adequate performance and reliability to support continued development. As a result, Labscape was re-implemented on *one.world* [5], a comprehensive runtime system that imposes a programming model specifically designed to support ubiquitous applications. The purpose of this paper is not to evaluate *one.world* against other systems for distributed programming; rather it is to offer some conclusions, based on our application experience, about the proper division of labor between the applications and systems communities in the development of ubiquitous computing environments.

In Section 2, we describe the user requirements of Labscape that place it squarely in the domain of ubiquitous computing, and which distinguish it from typical distributed applications. In Section 3, we review the system support alternatives that we had considered before deciding on *one.world*. Our purpose is to argue that our choice of *one.world* is reasonable, and that our experience demonstrates that many of the concepts in embodies are likely to be useful for a wide range of ubiquitous computing applications. In Section 4, we describe the architecture and the two implementations of Labscape, followed by a detailed comparison of the two implementations. In Section 5, we focusing on the implementation strategies we employed in both cases, and how they were driven by user requirements. We conclude in section 6 with summary of lessons learned.

2. The Application Requirements

An abstract flow-graph representation of a biology experiment is at the center of the Labscape user interface. The flow graph is a convenient abstraction for keeping track of planned and completed laboratory work. As laboratory work proceeds, the flow-graph is visually transformed into a record of the actual experiment, which is annotated with parameters used, observations made, and data files produced. Most importantly, the resulting electronic record can easily be shared or searched providing new opportunities for collaboration and communication. To achieve the high degree of adoption that is required for the resulting record to have significant value, it is important that the interface itself be designed to deliver benefits to the biologist in all phases of laboratory work. But, good interface design is not sufficient—our aim is for Labscape to be perceived as a basic laboratory utility like purified water, laboratory gasses, ventilation and electricity, which are all ubiquitously and available and easily accessed throughout the environment.

Figure 1 shows a map of a biologist's movements within the first ten minutes of an experiment in the pre-Labscape environment. This data was chosen to highlight some characteristics of laboratory work that strongly influence our design: 1) information access is highly interleaved with the physical activities required of the experiment; and 2) the biologists hands are typically full or busy when the they moves about the environment, and 3) information access is not currently well integrated into the laboratory workflow. Our goal is to build a system that improves the overall fluidity of the work process with respect to information access and recording.

In fact, workflows in a laboratory can be quite complex. Multiple biologists use the specialized workbenches of the typical laboratory, they each have multiple on-going experiments at any one time, and any one experiment can involve multiple biologists at different stages due to the highly specialized nature of some of the tasks.

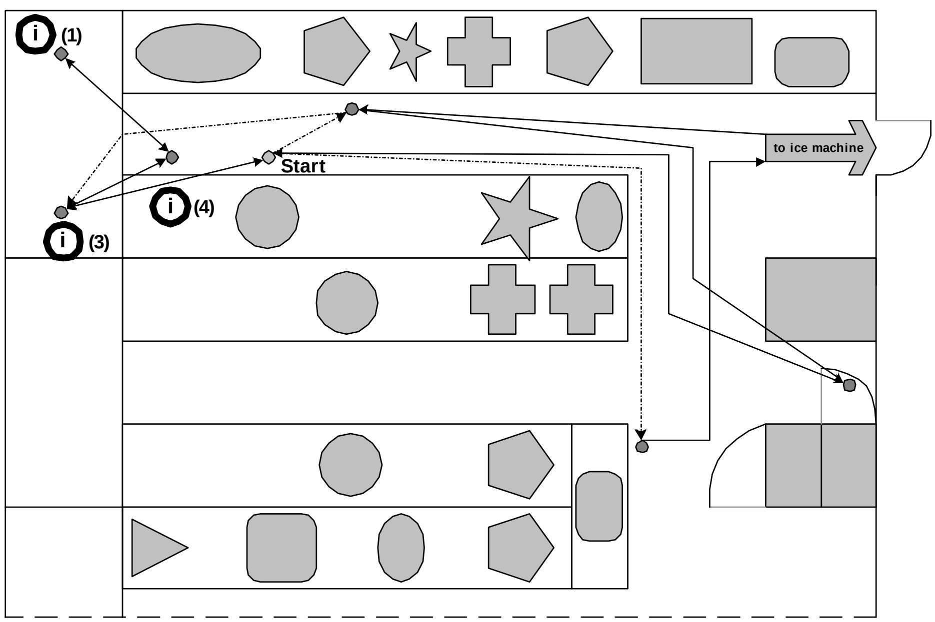

Figure 1: The first ten minutes of a biology experiment. Arrows represent (possibly multiple) movements between work areas. Solid and broken lines correspond to movements during which the biologists hands were busy or not respectively. The various shapes indicate different types of specialized work areas shared by biologists. The circled “i” symbol corresponds to locations where information was either accessed or recorded.

To increase the accessibility of information throughout the laboratory, without adding additional items for the biologist to carry, we have equipped the environment with several pen/touch tablet computers (Fujistu Stylistic 3400), and radio frequency ID (RFID) and barcode scanners at each of the work areas. Access to the Labscape application is controlled by a system that keeps track of user proximity to the work

areas. Proximity is detected either through direct touch interaction with the tablet computers, or through body-worn short-range infrared badges (like active badges [14]) that communicate with wireless sensors [15] arrayed along the lab bench. When our location system detects that the user has entered the region of influence of a device, such as a display, scanner, or digital camera, the user gains control over that region unless another user is currently present. If the device has a display, then the user's application interface appears, showing the evolving record of the current experiment.

Establishing the physical environment is, of course, the relatively easy part. Developing the software that orchestrates these devices into a *fluid* user experience is another matter. By fluidity, we mean that the tools and devices introduces into the lab should actually enhance the biologist's ability to focus on the biology. For Labscape, there are three major elements of a fluid user experience.

1. **Ubiquitous accessibility.** The biologists should be able to access their personal Labscape interface with low latency and high responsiveness at any work area that they visit.
2. **Flexible control of I/O resources.** Work areas in the laboratory are task specific rather than user specific. And, not all work areas will necessarily be served by a separate CPU. This means that control of devices must be granted to users based on location and context rather than on now those devices are physically connected to computers. This includes scanners and cameras as well as microphones, keyboards and specialized laboratory instruments.
3. **Robustness.** Our goal is that the biologists should not be concerned with file I/O, the risk of losing work due to system failures, or the need to be aware of state and topology of the network. This means that we must provide transaction-level persistence and support for disconnected operation.

Taken individually, these requirements are not unique. It is the combination of all three that is characteristic of ubiquitous computing, and which presents implementation challenges that go beyond traditional distributed systems. Take, for example, the dynamic nature of our network: machines may come and go from our system frequently since the pen tablet computers can easily be removed from their dock. This level of dynamism is not unusual; the set of clients connected to the Internet is highly dynamic. The fact that Labscape has computation spread across a large number of machines is also not unique. Thousands of machines are commonly employed to solve computationally intense problems like weather simulation. Finally the fact that Labscape has a variety of different types of components that interact in non-trivial ways is also not new. Most enterprise environments have a variety of component types including gateways and servers for web pages, files, and mail.

But, each of the above examples takes advantage of design trade-offs that simplify implementation in ways that are acceptable to their users, and their needs. The Internet works, despite its dynamics, in part because it has a simple interaction model that exposes delays and errors to the user. Parallel computations take advantage of regularity to simplify programming and deployment of large numbers of homogeneous interacting components. Finally, complex heterogeneous enterprise and supply chain systems are feasible in practice because the networks are stable—changes do occur, but they are carefully managed by skilled staff. In contrast, Labscape must allow for dynamic reconfiguration of the system in the presence of complex interactions between heterogeneous software and hardware components, without an army of support personnel, and yet, it must appear stable and reliable to its users.

Though we had a clear understanding of the user requirements from the outset, we did not have a clear understanding of what sort of systems support would be most useful in our case. For this reason, we first chose to develop the application entirely from scratch using networked file I/O for storage, and low-level networking APIs such as sockets [7] for communication. Though we were able to complete crucial user studies in a real user environment with this implementation, it was necessary for us to seek higher-level systems support for the next implementation so that we could focus our effort on application level concerns rather than detailed system level interactions. In Section 3, we outline the basis for our selection of *one.world*.

3. System Support for Ubiquitous Computing

Software engineers developing ubiquitous computing applications have a number of choices of platforms on which to build. This section compares and contrasts the programming models and capabilities offered by these platforms relative to *one.world*, our chosen platform.

One of the most common types of software platform for developing distributed applications are remote procedure call (RPC) systems. RPC systems have been around for decades [1] and have proven to be flexible and long-lived. Sun RPC is an example of a classic RPC system [2], more recent examples include XML/RCP [9] and SOAP [4]. Distributed object systems such as CORBA [6], Modula-3's network objects [3] and Java RMI [8] are the object-oriented counterparts of RPC systems. The fundamental benefit of these systems is that they allow developers to easily build simple distributed systems using familiar programming models, namely, procedure calls and object/method invocation. Ironically, while this transparency provides for a familiar programming model, it is what makes RPC systems a poor choice for building ubiquitous applications. Whenever the thread of control is executing remotely, the execution context of the caller is at the mercy of network connections that may be unreliable. Because the programming model intentionally hides the difference between local and remote execution, it is difficult to reason about the conditions under which a given component can safely execute. Furthermore, the inherently synchronous nature of RPC interactions limits the responsiveness of applications, as they need to wait for remote services to complete processing each procedure call. As a result, it is difficult to build robust and responsive ubiquitous applications using RPC systems. [1]

The problems with RPC are actually just symptoms of a more general issue: *ubiquitous computing environments are dynamic*. Thus, we have to find a way to write software components that respond appropriately to change. *one.world* addresses this problem in a general way: it insulates software components from changes that can be handled by the system, and it notifies them of changes that should be addressed at the application level. This, in turn, gives the application the opportunity to treat the user in a similar way. The application can insulate users from changes when possible, and interact with them when input is needed.

One.world is Java-based runtime system that executes on a standard JVM. Components are objects that are prohibited from accessing system resources directly. In the *one.world* programming model, RMI is prohibited, as are application level ownership and control of threads. Direct access to local resources such as the file system and the native operating system is discouraged but can be enabled by the developer. By restricting software components to consist of nothing but a collection of asynchronous event handlers, all interactions can be mediated by the *one.world* system. In this way, the system has the opportunity to handle changes or failures, and it can notify the component when necessary. Some examples of system notification events are activation and deactivation of components, event delivery failure, and relocation of a component from one node to another. By exposing change to the application, *one.world* allows the developer to determine how best to respond in terms of benefits to the user.

one.world provides alternatives that replace what is prohibited or restricted: a location independent tuple-store instead of file system access, event queues and timer events instead of threads, remote event passing instead of RMI, and a variety of other system-like utilities. By adhering to these alternatives, *one.world* guarantees that a component can execute on any node given enough physical resources. Because all interactions between components are through asynchronous events, *one.world* provides a rich event delivery infrastructure including early- and late-binding discovery, multicast, and lookups on the events themselves.

In addition, *one.world* allows components to be dynamically organized into hierarchical execution *environments*. Each environment contains a tuple-store for persistent data, and can contain running components and subordinate environments. The tuple-store can be used by components for data storage and for checkpointing of the execution state. Environments, not components, are the unit of migration in *one.world*, thus ensuring the availability of all state that is essential for continued execution after migration.

Our choice of *one.world* was due, in part, to the fact that it was developed and supported by our colleagues in the same computer science department. However, it is also clear that no other available system provided such a comprehensive set of features designed specifically to support ubiquitous

[1] Jini [10] provides much of what we want: a discovery service that enables dynamic composition and leases to control allocation of resources, and a discovery service to expose the dynamic set of services. But, because it is built on Java RMI, it suffers from the same limitations, described above, that apply to all RPC systems.

computing applications. Our primary concerns regarding the use of *one.world* were: 1) whether or not a real application, like Labscape, could be supported within the restricted programming model it requires; and 2) as a new system, *one.world* could well have had performance and reliability problems of its own. While the first issue has proved to be somewhat problematic, we have found the system to be largely stable and responsive.

Though the ensuing comparison paints a positive picture of our experience with *one.world*, we are comparing it only to our negative experience of having no systems support beyond basic sockets and distributed file systems. The point of this discussion is to argue, in general, for the role of systems support in developing ubiquitous applications, rather than to promote *one.world* as the best possible solution.

4. Comparison of two Implementations

The high-level component decomposition of the Labscape system remains largely unchanged between our two implementations, as does the primary method of composition—asynchronous events. In the first implementation, events are sent through socket connections between components. In the second, we rely on the late-binding discovery event delivery mechanism provided by *one.world*. Figure 2 shows the major components of Labscape. The following discussion describes each component and mentions important differences between the two implementations.

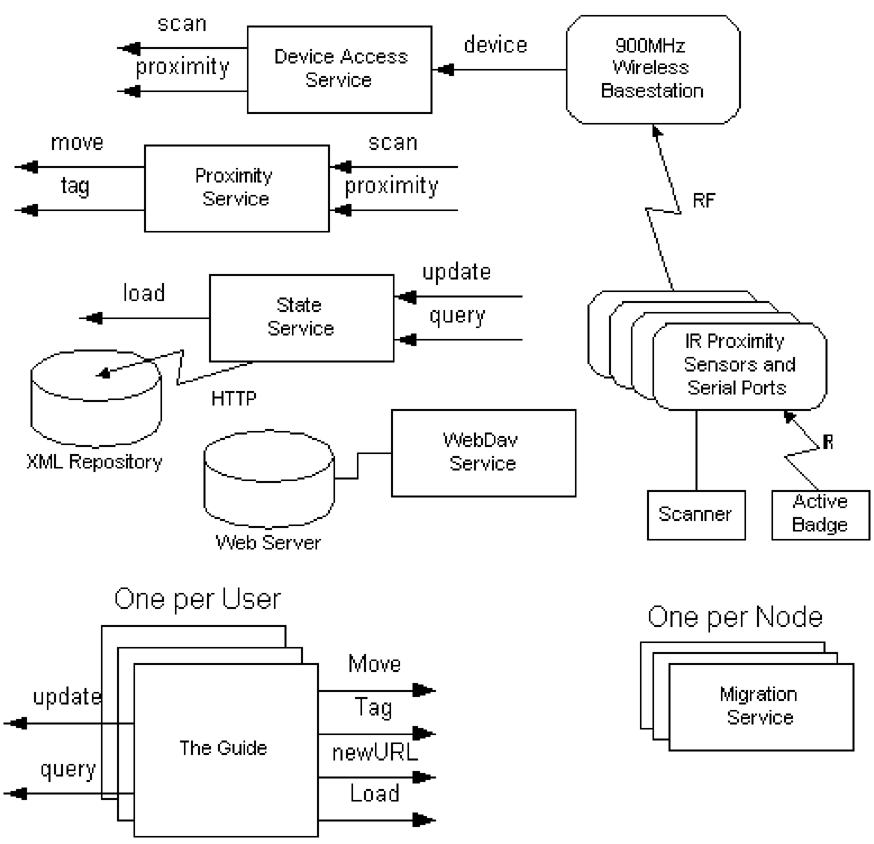

Figure 2. The major components of Labscape

The Guide is the main GUI component of the system**;** it presents the biologist with the flow-graph representation of current and past experiments. Incoming *move* events notify the Guide to migrate to a new display device. Incoming *tag* events and *newURL* events are data items that result from user interaction with the laboratory environment such as barcode scanning or creation of an image or data file. These data

items can pop up on the display for user interaction. As the user updates the experimental record, or asks for more information, the Guide emits corresponding *update* and *query* events. Incoming *load* events contain query responses but arrive asynchronously. The persistence oriented events, *update*, *query*, and *load*, are only part of the second implementation since the first relied directly on native file I/O.

The Device Access Service listens on the serial ports for incoming events from base stations that channel information from the wireless networked sensors. Such events can be of two types: *proximity* events and *scan* events. A *proximity* event contains the ID of the badge and of the sensor. The *scan* event contains the scanned data and the ID of the sensor. The incoming serial data is decoded, cast into appropriate *one.world* events, and then forwarded to the Proximity service via late-binding discovery.

The Proximity Service maintains associations between users and locations based on *proximity* events generated by the active badge detectors or user interactions with displays. When the Proximity service concludes that the user has moved to a new location, it emits a *move* event. The Proximity service also associates incoming *scan* events with users based on the sensor ID through which it was received. Incoming *scan* events are transformed into *tag* events and forwarded to the appropriate Guide through discovery.

The WebDAV Service watches specific directories in file system for files created by users, either directly or through the use of laboratory equipment. When a file appears in the user's directory, it is copied to a web server through a DAV interface, and it emits a *newURL* for that user. This is an example of a case in which the application requirements would not allow us to adhere to the *one.world* programming model.

The State Service, which exists only in our *one.world* implementation, provides a gateway to the long-term persistence engine, which is currently an XML database for representing laboratory procedures and data. The State service consumes *update* and *query* events and emits *load* events.

The Migration Service, which exists only in our original implementation, listens for incoming object streams from migrating Guide components, and maintains a control connection with the proximity service. The Migration service must always be running on each CPU in the system. In our second implementation, the *one.world* runtime environment eliminated the need for this service.

In our first development effort, we knew that we could not address all three of the fluidity requirements described in Section 2. Instead, we focused our efforts on ubiquitous accessibility and flexible control of I/O resources, as these were more important than robustness for evaluation of the user interface. The infrastructure that we built consists of ~3000 non-commenting source statements (NCSS) that required three full-time developers about three months to produce. This NCSS count does not include the application-specific components (mostly user interface and flow-graph objects), which consist of another ~7700 NCSS. Despite our efforts, this implementation fell short in terms of both ubiquitous accessibility and flexibility, and even more than expected in terms of robustness. Refer to Section 5 for the details as to how and why.

The second implementation of Labscape, using *one.world*, required two skilled developers approximately two months to complete. One of the developers was also on the development team for the sockets implementation. This effort included porting the main application specific component (the Guide) and re-developing all of the other components to conform to the *one.world* programming model. As before, the code base can be split into two parts: the application-specific components consisting of ~9700 NCSS and the infrastructure components, requiring ~4700 NCSS. Though there is more code in the *one.world* infrastructure components than in the sockets-based infrastructure components, there are three important differences that are not reflected in these numbers:

- The *one.world* version has more functionality in both the GUI and in the infrastructure, as well as increased stability and performance. The extra functionality in the current *one.world* version is primarily for transaction-level persistence.
- The *one.world* code is at same level as the application ontology—expressed as event types and event handlers. Though bulky, it is easier code to understand and maintain than the lower-level programming found in the sockets-based implementation.
- The infrastructure components in the *one.world* implementation are generic and independent enough to be used by other applications running in the same environment. This cannot be said of the original effort.

These are important properties of the *one.world* implementation that demonstrate its benefits for application developers: higher-level abstractions and greater code reusability result in more robust

applications. Though the two architectures are very similar, the return on our development effort was quite different with respect to our fluidity goals. The reasons for these differences are discussed in detail below.

5. Evaluation

In this section, we evaluate, in detail, the two separate implementations with respect to the three user requirements defined in Section 2: ubiquitous accessibility, flexible control of I/O resources, and robustness.

5.1 Ubiquitous Accessibility

We have considered three alternative strategies for delivering the application interface to the touch-tablet computers scattered throughout the environment: virtual terminal, on-demand state migration, and proactive state replication. These alternatives have distinctly different characteristics with respect to latency, responsiveness, resource utilization, and robustness. Virtual terminal technologies such as X-windows, Microsoft Terminal Server and VNC offer relatively low initial latency and low local resource requirements in exchange for high network bandwidth, poor or variable responsiveness, and no support for disconnected operation. State migration suffers from higher initial latency and increase use of local resources, but it offers better response times and can run disconnected after migration. Proactive replication achieves very low latency in exchange for memory capacity and network bandwidth to maintain and synchronize each copy. Choosing the right approach is a legitimate application level concern.

Hoping to keep it simple, we first implemented a VNC strategy for interface migration. Though initial latency under VNC was adequate, our users considered the responsiveness of the UI to be unsatisfactory, even over a reliable 100Mb local area network. This was primarily due to our extensive use of pop-up windows and menus in addition to smooth animation in the movement of screen objects. To improve responsiveness, we implemented a state migration scheme over TCP sockets that works as follows:

- The Proximity service sends a *move* event to the Guide when the user's presence is detected at a new work area. The *move* event includes the IP address and port number of the Migration service running at the destination work area.
- Upon receipt of a *move* event, the Guide releases local resources (files, socket connections, database connections, etc); serializes itself through a new socket connection to the destination Migration service; and then terminates locally.
- The newly instantiate Guide re-establishes needed socket connections and file descriptors and then resumes execution.

This scheme worked after a significant development effort, but we suffered from both reliability and performance problems. To attack the performance issues, we carefully identified the minimal amount of application state that must be preserved to provide the user with a strong sense of continuity. In the end, we left all of the GUI components behind in favor of rebuilding them on demand at the new location. Unfortunately, our migration system remained unreasonably slow. Our reliability problem was related to migration interaction with the Java Swing event loop. Though we could have put further effort into diagnosing these problems and optimizing our migration scheme, our need to continue with user evaluation and functional development was more pressing, and provided strong motivation to move to the *one.world* platform. There were two major lessons form this experience:

- It is natural in traditional application development to be somewhat careless about creation and use of state. This includes both open and closed files in the file system, network connections, data base connections, and the execution context of every thread in a multi-threaded system. As a result, state migration is difficult to add to an existing system, and one must continuously reconsider the effects of migration when adding new features. In our case, the addition of a simple utility that logged user interactions to a file created a number of migration bugs that had to be painstakingly isolated.

- Successful realization of a state migration mechanism is beyond the scope of what application developers should do. Application developers should definitely control *what* bits to move and *when*, but they should have to be concerned with *how* to move the bits. Systems support should offer several methods with associated performance and reliability guarantees, and it should notify the application about changing conditions that might affect the choice.

The restricted programming model of *one.world* directly addresses the first issue by encapsulating all essential state within environments. And, since *one.world* components are prohibited from creation and ownership of threads, an environment can be safely serialized and migrated once all of its components' event handlers have completed execution. As application developers, our only responsibility was to declare soft state as transient and fill in template methods for activation and deactivation. Since environment migration is a core utility of *one.world* that runs on every node of the network, there was no need for us to build our own migration service. This implementation realized a 5-10x decrease in migration latency, eliminated intermittent migration failures, and allowed us to completely eliminate our own custom migration service.

Table 1 shows the migration data sizes and times for three flow-graphs of different sizes on the *one.world* implementation. The reported times are averages of hundreds of migrations. A 64-sample flow-graph contains approximately 640 nodes and 1200 edges, and represents a large experiment. The migration times are measured from the receipt of the move event to the time that the complete GUI becomes available at the destination. Because the data sizes are small, most of the time is spent in deserialization and reconstruction of the GUI components, which were not serialized, rather than in data transmission.

Data Structure Size	Size of Migrating Data Stream	Average Time to Complete Migration
Empty Guide Migration	19.5 Kbytes	2448ms
32 Sample Guide Migration	174.8 Kbytes	4476ms
64 Sample Guide Migration	323.0 Kbytes	7089ms

Table 1: Data set sizes and migration times.

There is still a possibility that these migration latencies could become unacceptable. If current user studies bear this out, then we will have to implement a more proactive strategy. Currently, this would require us to implement a replication layer on top of *one.world*'s existing tuple-store. Fortunately, replication is a focus of a new set of services being developed within *one.world*. Though perhaps we could get direct support for replication from another storage subsystem now, it likely would not conform to the *one.world* programming model. This is one of the risks of working in a restrictive system.

5.2 Flexible Control of I/O Resources

The application requirement is to allow users to access I/O devices, such as keyboards, cameras, laboratory instruments, and barcode scanners, based on context and location regardless of how the device is physically connected to the system. Flexibility in our first implementation was realized by tearing down and rebuilding socket connections between Guide components and proxies for bench-top I/O devices. This constant reconfiguration was accomplished by maintaining permanent control connections between the Proximity service and each user's Guide component. Predictably, this approach led to serious scalability and robustness problems.

The scalability problem stems from our need to maintain large numbers of active socket connections. Every open socket requires operating systems resources. While there might be system level solutions to this resource utilization problem that would still allow large numbers of open sockets, such solutions should be beyond the concern of application developers. The robustness problem stems from our reliance on long-term socket connections without extensive failure detection and recovery programming. Once again, this problem can be solved with careful implementation of low-level protocols that should not be required of application developers. At its zenith, our sockets-based system had to be completely rebooted after any single component failure. The mean time between failures (MTBF) was about 30 minutes with just four work areas under light user light load.

The second implementation avoids the scalability and robustness problems of the first through the use of *one.world*'s communication infrastructure. By using asynchronous event delivery through late-binding discovery, resource utilization scales with activity rather than with the number of components. And, because we do not maintain any permanent connections, failures in the system tend to be transient rather than permanent. Though we still have to deal with the potential for lost events, this can be done selectively, only in the cases where some action is necessary. For example, no response is necessary for lost *proximity* events because they represent transient information that will soon be refreshed, whereas the user should be notified in the case that the system could not deliver a barcode *scan* event.

The Proximity service plays essentially the same role in both our sockets and *one.world* implementations. But, instead of maintaining hundreds of open socket connections, the new Proximity service simply exports one event handler of the right type. Our current implementation has to process an average load of 170Kbytes/sec of incoming proximity event data to support approximately 100 users each generating one *proximity* event per second. Because this is an entirely manageable load, the Proximity service can now be viewed as a utility that can be shared by many applications running in the same environment, instead of as a private component of the Labscape system.

The background load on the Proximity service for the above scenario is shown in Figure 2. Network activity was logged using packet sniffers installed at the node running the Proximity service. Because we were simulating stationary users, the only outgoing events are for the periodic *move* notifications sent by the Proximity service to refresh the Guides. The initial bursts of activity correlate to some start-up traffic for registration and acknowledgement of the 100 simulated users, which occurred in two batches.

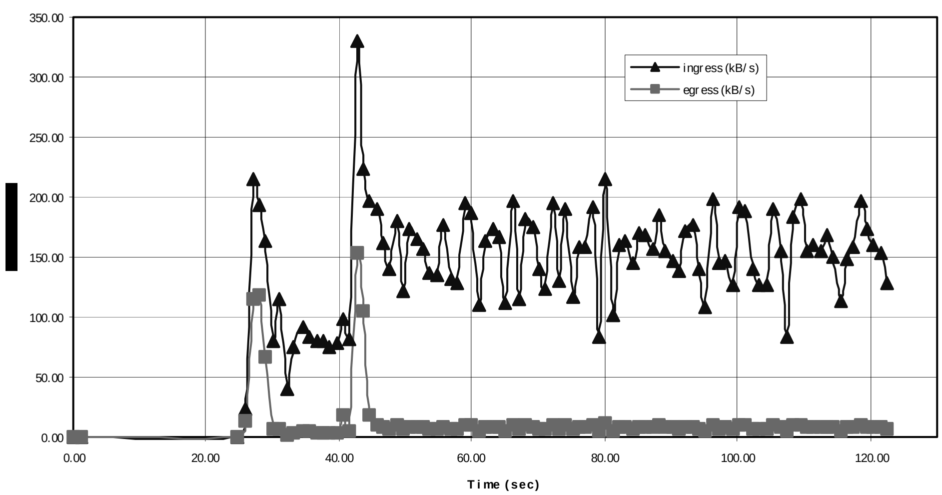

Figure 3: Background workload for the Proximity service

There was one useful property of sockets that we lost when we moved to *one.world*: in-order delivery of events. In some cases order is not important, such as for the IR *proximity* events. But order is essential to the user in barcode scanning. Lack of direct support for ordered event semantics placed an unexpected burden on the application development team. This issue is discussed in more detail in the next section.

5.3 Robustness

In Section 2, we listed two aspects of robustness that are important to our users: transaction level persistence, and disconnected operation. Of these, we have only addressed persistence in our current *one.world* implementation as we expect this to have the largest initial impact on the user's sense of fluidity and security.

For persistence, we implemented a transaction level interface to an XML database to eliminate the need for user awareness of file I/O. It was important for our implementation not to allow the responsiveness of the interface to suffer from database transaction delays. This goal was achieved by creating a State service that could accept *update* and *query* events, expressed as XML strings, from the Guide. The State service is the gateway that interacts with the external XML database system through HTTP connections. In response, to *query* events, the State service emits *load* events containing information returned by the XML repository. The asynchrony of the interaction is the primary reason that we were able to achieve our combined goal of incremental persistence and high responsiveness. But, because *one.world* prohibits the use of application-level threads, the programmer must explicitly maintain enough context for proper processing of the eventual response or timeout event. This requirement is an inconvenience for the developers and can result in code that is difficult to trace and understand.

Two other problems that we had to address at the application level was the possibility of out-of-order event delivery and lost events that could be devastating to the integrity of the persistent experiment record. The solution to the first problem was to maintain a reordering buffer in our State service for each active Guide. The solution to the second problem was to make use of *operation* primitives provided by *one.world*. Operations are just like regular events, except that they are guaranteed to receive exactly one (asynchronous) response. If the response is an acknowledgement, then the event has safely arrived at the destination. However, if the response is a timeout, then the event may or may not have arrived. Thus, a resend could result in a duplicate event arriving at the destination. We solved this problem by having our reordering buffer reject such duplicates.

The need for this pattern has proved to be the rule rather than the exception in our system. The *proximity* event is the only one in our system that is naturally insensitive to order and duplication. Though the *one.world* operation primitive proved to be useful, more direct support from the system for ordering and idempotency would have kept our application developers from implementing their own, likely imperfect, protocols. Several patterns like these are providing the *one.world* developers with potential enhancements to their system.

Figure 4 shows the incremental network activity generated by a single user, including migration, persistence events, and proximity activity. Three computers were used for this experiment, one of which was running the infrastructure services. The other two represent work areas. The user starts the Guide component with a 64-sample flow-graph and then migrates once to each work area. At the second work area, the user interacts with the GUI, causing *update* events to be sent to the State service. At an average total load of 50Kbytes/sec per user, a single 100BaseT subnet should be able to support at approximately 60 users before overloading the network (which we consider to occur at utilizations greater than 33%). Since one biologists requires approximately 150 square feet of space, our basic system should be able to support approximately 9,000 square feet of laboratory space.

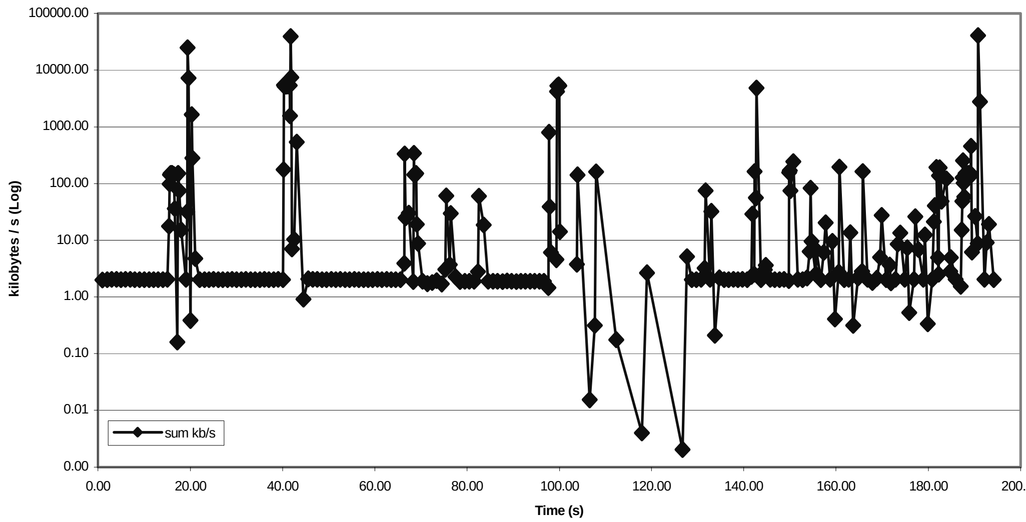

Figure 4. Incremental network traffic associated with a single user (note log scale).

In the near future we plan to extend the system to further increase the robustness of Labscape. Two examples are support for disconnected operation and service redundancy. One solution to disconnected operation is to ensure that all essential components and state are always available. The environment hierarchy can be used to make this guarantee. As an example, Labscape has a minor component, called a Generator, that creates new experiment plans from a predefined library of laboratory procedures, and then sends them to the State service in an *update* event. Currently the Guide has to fetch these new plans form the State service in a separate *load* request. This model does not bode well for disconnected operation. But, by subordinating the Generator to the Guide, newly generated plans contained in *update* events can be snooped by the Guide on their way to the State service. This approach is both more robust and provides faster response to the user. The power of the *one.world* hierarchical environment abstraction is evident in that the Generator does not have to be modified at all to handle this interaction. Though an additional handler must be added to the Guide to implement the snooping, it will work with any component that generates *update* events, not just the Generator. Furthermore, the Generator does not have to handle issues that arise when a State service is unreachable since the Guide will have to take care of the problem in the manner described below.

When the Guide receives failure responses for *update* operations sent to an unreachable State service, the updates themselves can be written to the Guide's local tuple-store. Once the State service comes back on-line, the pending updates can be resent and removed from the tuple-store. Because the tuple-store is part of the environment, no extra development work is required to cope with migration and checkpointing in the presence of disconnected operation. This is an example of the orthogonality that we have found to be very important in the development of this complex distributed system.

Redundancy is an important aspect of robustness. Though it will be easy for our team to develop basic redundant systems through the use of *one.world*'s multicast and anycast delivery options, the system could provide even more assistance. Our job, as developers, should be only to ensure that the services we write can be replicated without adverse interactions. This can be achieved by designing services that are either stateless, or that rely only on soft-state for which synchronization is not necessary. This is already the case in Labscape for all but the Guide and State services. The role of the system, on the other hand, should be to balance load through event delivery, and placement and replication of services. These two capabilities are not currently provided by *one.world*, except to a limited degree, for the internal discovery service.

5.4 A User Scenario

It is common for real laboratory work to cross physical and organizational boundaries for access to special equipment, facilities, or biochemistry that cannot easily be reproduced. The following realistic scenario highlights how we imagine that ubiquitous access, flexibility, and robustness can work together to provide a fluid user experience even in the wide area. And, it raises a potential maintenance and interoperability problem associated with our system.

Before leaving her home environment, the biologist would migrate her plan for the day to her PDA (just the Guide's environment). Given sufficient resources, the Guide could run stand-alone on the PDA, allowing the biologist to view and modify her plan en-route.

Upon arrival, her PDA joins the local network and collaborates with the local *one.world* nodes to synch up the discovery services. This is an automatic procedure that is already supported by *one.world*. As long as it is the allowed by the host facility, her Guide can now migrate to any of the host laboratory's display resources and access devices through that environment's proximity system. *If the Labscape event ontology is respected*, then everything works as it did in the home environment, except for perhaps one thing: the local State service should reject the user's database *update* requests because she does not have permission, and any attempt for those events to reach the home state service might be blocked by a firewall. Yet, through checkpointing and update backups in the tuple-store, the Guide can continue to operate and cache the work locally as it migrates around the host's facility.

Upon departure, the Guide is pulled back to the user's PDA, and the pending updates are committed as soon as the home State service becomes accessible again. It is no problem if the PDA runs out of batteries on the way home, as periodic checkpointing will ensure that the work is saved in the device's nonvolatile storage. This scenario exemplifies how migration, flexibility, and robustness should work together to deliver a fluid user experience. But it also points out how our application is susceptible to incompatibilities between the ontologies of different versions of the same application, or between completely different applications that share services in the same environment.

In the end, we did not completely succeed in obeying all of restrictions of *one.world*'s programming model. We still rely on access to the file system to detect when data files are created by the user through interactions with legacy laboratory equipment, and we still need to make native system calls for external utilities such as the third-party graph layout tool that we rely on in our GUI. To the extent that we violate environment encapsulation, *one.world* cannot provide its portability and change-notification guarantees. It is entirely reasonable that, in such cases, the responsibility falls back to the application developer to ensure that state is properly managed at activation and deactivation time. Most of these special cases could have been eliminated if *one.world* offered a standard wire protocol for exchanging asynchronous events with external components.

6. Status and Conclusions

The current *one.world* based implementation of Labscape has sufficient performance and robustness to meet the needs of our users for authentic evaluation of the user experience. We are currently engaged in a formal user study that will assess the impact that Labscape has on the overall fluidity of the laboratory workflow. But most importantly, we are confident that we can continue to maintain and extend this application into the future while we learn more about how *one.world*'s guarantees can provide unique capabilities. In the future, we expect to have several applications running over the same set of basic services. It may well be that some of the services we have created for Labscape will be so universal that they will be incorporated into the core system just as discovery already has been. The following three major themes have emerged in our experience with two implementations of Labscape.

1. Minor failures should not become major failures. In our original implementation, a lost socket connection resulted in the complete loss of the system. But, a lost socket connection is a minor failure that should be recoverable. Insulating users from minor failures requires sophisticated algorithms operating at a fairly low level. These algorithms should be packaged and made available to developers in the form of system support. In fact, *one.world* uses sockets to move events, but this is complete hidden from the application components, which only see asynchronous message delivery. If the system can't

solve the problem, then it should notify the application components. As an example, a warning to the user might the most appropriate response to a application whose serialized footprint has grown to large for safe checkpointing on the current device.

2. Diversity and orthogonality are essential. The key difference between the two development efforts was that, in the *one.world* case, we were able to address each of our concerns independently. Ideally, systems that support ubiquitous computing should provide several alternative mechanisms that can be mixed and matched with predictable results. As an example, once our migration system was working, it never failed due to some other feature that we added to the system, such as when we added transaction-level persistence. This is counter to our experience with the first system, in which the addition of new features frequently had adverse interactions with existing capabilities.
3. An external asynchronous event wire protocol is needed. *one.world* presents a restrictive programming model in exchange for certain guarantees on portability and adaptability that we have in fact found to be important for our application. But, not all components in the system should be forced to make this same tradeoff. In the case of Labscape, the webDAV and Device Access services both need to access native system resources and should run in the native environment. But, we were prevented from implementing such a hybrid system because *one.world* did not offer an external event interface.

Labscape running on *one.world* is an experiment into how applications can be made more adaptive to changes in their execution environment. The biggest surprise to our development team was the relative ease with which we could adhere to the seemingly restrictive programming model and how much it helped us build a fluid system. The fact that *one.world* was not designed specifically with Labscape in mind supports the idea that ubiquitous applications have many requirements in common. Our experience is that such applications are indeed facilitated by asynchronous event models, careful delineation of essential volatile and non-volatile state, and runtime support for appropriate handling of changes and failures that arise in dynamic environments.

Acknowledgements

We would like to acknowledge the NSF/REC, The DARPA Expedition into Ubiquitous Computing, Intel Research, Intel Corporation Research Council.

References

[1] Birrell, A. D., Nelson, B. J., "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems*, vol. 2, no. 1, pp. 49-59, Feb, 1984.

[2] Srinivasan, R., *Remote Procedure Call Protocol Specification Version 2*, Internet Engineering Task Force, RFC No. 1831, Augm 1995, http://www.ietf.org/rfc/rfc1831.txt?number=1831

[3] Birrell, A., Nelson, G., Owicki, S., Wobber, E., Network Objects, Technical Report, Digital Equipment Systems Research Center, #115, Palo Alto, California, Feb, 1994

[4] Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Frystyk, N., Thatte, S., Winer, D., Simple Object Access Protocol (SOAP) 1.1, World Wide Web Consortium Note, Cambridge Massachusetts, May, 2000.

[5] Grimm, R., Davis, J., Lemar, E., Macbeth, A., Swanson, S., Gribble, S., Anderson, T., Bershad, B., Borriello, G., Wetherall, D., Programming for Pervasive Computing Environments, University of Washington Technical Report UW-CSE-01-06-01, June, 2001.

[6] Siegel, J., CORBA Fundamentals and Programming, John Wiley & Sons, April, 1996, ISBN 0-471-12158-7.

[7] Stevens, R. W., UNIX Network Programming, Prentice Hall, vol. 1, 2nd edition, 1998, ISBN 0-13-490012-X

[8] Sun Microsystems Corporation, Java Remote Method Invocation Specification, Rev 1.7, Palo Alto, California, December 1999.

[9] Winer, D., XML-RPC Specification, http://www.xmlrpc.com/spec, UserLand, Burlingame California, October 1999.

[10] Arnold, K., O'Sullivan, B., Scheifler, R.W., Waldo, J., Wollrath, A., *The Jini Specification*, Addison Wesley, 1999, ISBN 0-201-61634-3.

[11] Oki, B., Pfluegl, M., Siegel, A., Skeen, D., "The Information Bus – An Architecture for Extensible Distributed Systems", *SOSP14*, pp 58-68, Dec, 1993.

[12] Cugola, G., Di Nitto, E., Fuggetta, A., "Exploiting and Event-Based Infrastructure to Develop Complex Distributed Systems", *Proceedings of the ACM 1998 international conference on Software Engineering* , Kyoto, Japan, 1998

[13] Corriero, N., Gelernter, D., "Linda in Context", *Communications of the ACM*, 32,4, April, 1989.

[14] Want,R., Hopper, A., Falcao, V., Gibbons, J., *The Active Badge Location System*, Technical Report 92.1, 1992, ORL, 24a Trumpington Street, Cambridge, CB2, 1QA.

[15] Hill, J., Szewczyk, R., Woo, A., Hollar, W., Culler, D., Pister, K., "System architecture directions for network sensors", ASPLOS 2000.